GPU Computing with CUDA

Dan Negrut

Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
University of Wisconsin-Madison

Milano
10-14 December 2012
Before we get started…

- Yesterday: CUDA basics
  - How to launch a kernel
  - How to define an execution configuration (number of blocks and threads/block)
  - CUDA API: how to copy data back and forth from host to device

- Today:
  - Get familiar with the memory hierarchy on NVIDIA’s GPUs
  - Understand the scheduling of threads for execution on an SM
  - Issues related to writing effective CUDA code
    - Warp divergence, use of shared memory, etc.
  - Conclude with another hands-on session: focus on use of shared memory
Memory Wall
Memory Wall

- Memory Wall: What is it?
  - The growing disparity of speed between the chip and performing off-chip memory transactions

- Memory latency is a barrier to performance improvements
  - Current architectures have ever growing caches to improve the “average memory reference” time to fetch or write instructions or data

- Memory Wall: due to latency and limited communication bandwidth beyond chip boundaries.
  - From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory access speed only improved at 10%
Memory Bandwidths
[typical embedded, desktop and server computers]

- CPU
  - Registers
  - Register reference: Size: 500 bytes, Speed: 250 ps

- Cache
  - Cache reference: Size: 64 KB, Speed: 1 ns

- Memory
  - Memory reference: Size: 1 GB, Speed: 100 ns

- I/O bus
  - I/O devices
    - Disk memory reference: Size: 1 TB, Speed: 10 ms

© 2007 Elsevier, Inc. All rights reserved.
Memory Bandwidths
[typical embedded, desktop and server computers]
Memory Bandwidths
[typical embedded, desktop and server computers]
Memory Bandwidths
[typical embedded, desktop and server computers]
Memory Bandwidths
[typical embedded, desktop and server computers]

- CPU
  - Registers
  - Register reference
    - Size: 500 bytes
    - Speed: 250 ps

- Cache
  - Cache reference
    - Size: 64 KB
    - Speed: 1 ns

- Memory
  - Memory reference
    - Size: 1 GB
    - Speed: 100 ns

- I/O bus
  - I/O devices
    - Disk memory reference
      - Size: 1 TB
      - Speed: 10 ms

© 2007 Elsevier, Inc. All rights reserved.
Memory Speed:
Widening of the Processor-DRAM Performance Gap

- The processor: Moving so fast that it left the memory far behind
  - The CPU constantly dragged down by sluggish memory

- Plot on next slide shows on a *log* scale the increasing gap between CPU and memory speeds

- The memory baseline: 64 KB DRAM in 1980

- Memory speed increasing at a rate of approx 1.07/year
  - However, processors improved
    - 1.25/year (1980-1986)
    - 1.52/year (1986-2004)
    - 1.20/year (2004-2010)
Memory Speed:
Widening of the Processor-DRAM Performance Gap

![Graph showing the widening performance gap between processor and memory over time from 1980 to 2010.](image)

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Memory Latency vs. Memory Bandwidth

- Latency: the amount of time it takes for an operation to complete
  - Measured in seconds
  - The utility “ping” in Linux measures the latency of a network
  - For memory transactions: send 32 bits to destination and back, measure how much time it takes! gives you latency

- Bandwidth: how much data can be transferred per second
Latency vs. Bandwidth Improvements Over the Last 25 years

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
The Memory Ecosystem
[NVIDIA cards specific]

- The memory space is the union of
  - Registers
  - Shared memory
  - Device memory, which can be
    - Global memory
    - Constant memory
    - Texture memory

- Remarks
  - The constant memory is cached
  - The texture memory is cached
  - The global memory is cached only in devices of compute capability 2.X

- Mem. Bandwidth, Device Memory:
  - Approx. 140 GB/s
GPU: Underlying Hardware
[Tesla C1060]

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)
  - Look closer…
    - You do see shared memory on the SM
    - You don’t see global memory on the SM
GPU: Underlying Hardware
[Tesla C1060]

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)
  - Look closer…
    - You do see shared memory on the SM
    - You don’t see global memory on the SM
CUDA Device Memory Space Overview
[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on a SM on Tesla C1060

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memory
CUDA Device Memory Space Overview
[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on a SM on Tesla C1060

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memory

IMPORTANT NOTE: Global, constant, and texture memory spaces are **persistent** across kernels called by the same host application.
Global, Constant, and Texture Memories
(Long Latency Accesses by Host)
Global, Constant, and Texture Memories (Long Latency Accesses by Host)
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
Global, Constant, and Texture Memories
(Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between *host* and *device*
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads
The Concept of Local Memory
[a misnomer]

- Note the presence of local memory, which is virtual memory
  - If too many registers are needed for computation (“high register pressure”) the ensuing data overflow is stored in local memory
  - “Local” means that it’s local, or specific, to one thread
  - In fact local memory is part of the global memory
  - Long access times for local memory (in Fermi, local memory might be cached)
Memory Space, Tesla C1060
[Compute Capability 1.3]

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A - resident</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A - resident</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

- BTW, off-chip still means that’s on the device, but nonetheless this translates into slow access time
- NOTE: Fermi caches local memory, as well as global memory data transactions
Access Times  [Tesla C1060]

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached
Matrix Multiplication Example, Revisited

- **Purpose**
  - See an example where one must use *multiple blocks of threads*
  - Emphasize the role of the *shared memory*
  - Emphasize the need for the `_syncthreads()` function call

- **NOTE:** use same one dimensional array to store the entries in the matrix
  - Drawing on the *Matrix* data structure discussed yesterday
Why Revisit the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix $M$ (its entries) is copied from global memory to the device $N$.width times
  - The matrix $N$ (its entries) is copied from global memory to the device $M$.height times

- When solving a numerical problem the goal is to go through the chain of computations as fast as possible
  - You don’t get brownie points moving data around but only computing things
A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

- An advantageous way of performing computation on the device is to partition ("tile") data to take advantage of fast shared memory:
  - Partition data into data subsets (tiles) that each fits into shared memory
  - Handle each data subset (tile) with one thread block by:
    - Loading the tile from global memory into shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory back to global memory
Multiply Using Several Blocks

- One block computes one square sub-matrix $C_{sub}$ of size $Block\_Size$

- One thread computes one entry of $C_{sub}$

- Assume that the dimensions of $A$ and $B$ are multiples of $Block\_Size$ and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
  - In this example work with $Block\_Size=16\times16$
Multiply Using Several Blocks

- One **block** computes one square sub-matrix \( C_{\text{sub}} \) of size \( \text{Block}_\text{Size} \).
- One **thread** computes one entry of \( C_{\text{sub}} \).

Assume that the dimensions of \( A \) and \( B \) are multiples of \( \text{Block}_\text{Size} \) and square shape:

- Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest.
- In this example work with \( \text{Block}_\text{Size} = 16 \times 16 \).

NOTE: Similar example provided in the CUDA Programming Guide 4.2.
A Block of 16 X 16 Threads
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C) {
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // (continues with next block…)
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    (continues with next block…)
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMemcpy((void**)&Ad, A, size, cudaMemcpyHostToDevice);

    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMemcpy((void**)&Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMemcpy((void**)&Cd, C, size, cudaMemcpyDeviceToHost);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}

(continues below…)

(continues with next block…)
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
}
Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Shared memory for the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Shared memory for the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep) {

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();

        // Write the block sub-matrix to global memory;
        // each thread writes one element
        int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        C[c + wB * ty + tx] = Csub;
    }
}

// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
  // Block index
  int bx = blockIdx.x; // the B (and C) matrix sub-block column index
  int by = blockIdx.y; // the A (and C) matrix sub-block row index

  // Thread index
  int tx = threadIdx.x; // the column index in the sub-block
  int ty = threadIdx.y; // the row index in the sub-block

  // Index of the first sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;

  // Index of the last sub-matrix of A processed by the block
  int aEnd = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;

  // Index of the first sub-matrix of B processed by the block
  int bBegin = BLOCK_SIZE * bx;

  // Step size used to iterate through the sub-matrices of B
  int bStep = BLOCK_SIZE * wB;

  // The element of the block sub-matrix that is computed
  // by the thread
  float Csub = 0;

  // Shared memory for the sub-matrix of A
  __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

  // Shared memory for the sub-matrix of B
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

  // Loop over all the sub-matrices of A and B required to
  // compute the block sub-matrix
  for (int a = aBegin, b = bBegin;
   a <= aEnd;
   a += aStep, b += bStep) {
    // Load the matrices from global memory to shared memory;
    // each thread loads one element of each matrix
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    __syncthreads();

    // Multiply the two matrices together;
    // each thread computes one element
    // of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }

  // Write the block sub-matrix to global memory;
  // each thread writes one element
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Shared memory for the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Shared memory for the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep) {
        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
Synchronization Function

- It’s a device lightweight runtime API function
  - `void __syncthreads();`

- Synchronizes all threads in a block (acts as a barrier for all threads of a block)

- Once all threads have reached this point, execution resumes normally

- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory

- Allowed in conditional constructs only if the conditional is uniform across the entire thread block
The Three Most Important Parallel Memory Spaces

- **Register:** per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit if not cached)
- **Shared Memory:** per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication
- **Global Memory:** per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication
SM Register File (RF) [Tesla C1060]

- Register File (RF)
  - 64 KB (16,384 four byte words)
  - Provides 4 operands/clock cycle
  - Note: typical CPU has less than 20 registers per core

- TEX pipe can also read/write RF
  - 3 SMs share 1 TEX

- Global Memory Load/Store pipe can also read/write RF

See Appendix F of the CUDA Programming Guide for amount of register memory available on different compute capabilities
Programmer View of Register File

- Number of **32 bit** registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla C1060
  - 32K on Tesla C2050

- Size of Register File dependent on your compute capability, not part of CUDA

- Registers are *dynamically partitioned* across all Blocks assigned to the SM

- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks

- A thread in a Block can only access registers assigned to itself

Possible per-block partitioning scenarios of the RF available on the SM
Matrix Multiplication Example
[Tesla C1060]

- If each Block has 16x16 threads and each thread uses 20 registers, how many threads can run on each SM?
  - Each Block requires 20*256 = 5120 registers
  - 16,384 = 3 * 5120 + change
  - So, three blocks can run on an SM as far as registers are concerned

- What if each thread increases the use of registers from 20 to 22?
  - Each Block now requires 22*256 = 5632 registers
  - 16,384 < 16896= 5632 *3
  - Only two Blocks can run on an SM, about 33% reduction of parallelism!!!

- Example shows why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism
  - One way to find out how many registers you use per thread is to invoke the compile flag `-ptax-options=-v` when you compile with nvcc
More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each, or run a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can trade off between instruction-level parallelism and thread level parallelism
    - TLP: many threads are run
    - ILP: few threads are run, but for each thread several instructions can be executed simultaneously

Constant Memory

- This comes handy when all threads use the same *constant* value in their computation
  - Example: $\pi$, some spring force constant, $e=2.7173$, etc.

- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)

- A constant value can be broadcast to all threads in a warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block
  - When all threads in a warp read the same constant memory address this is as fast as a register
Example, Use of Constant Memory
[For compute capability 2.0 (GTX480, C2050) – due to use of “printf”]

```c
#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample() {
    float circum = 2.0f*dansPI*threadIdx.x;
    printf("Hello thread %d, Circ=%.2f\n", threadIdx.x, circum);
}

int main(int argc, char **argv) {
    float somePI = 3.141579f;

    cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
    myExample<<<1, 16>>>();
    cudaThreadSynchronize();

    return 0;
}
```
Example, Use of Constant Memory
[For compute capability 2.0 (GTX480, C2050) – due to use of “printf”]

#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample() {
    float circum = 2.f*dansPI*threadIdx.x;
    printf("Hello thread %d, Circ=%5.2f\n", threadIdx.x, circum);
}

int main(int argc, char **argv) {
    float somePI = 3.141579f;

    cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
    myExample<<<1, 16>>>();
    cudaThreadSynchronize();
    return 0;
}
Memory Issues Not Addressed Yet…

- Not all global memory accesses are equivalent
  - How can you optimize memory accesses?
    - Very relevant question

- Not all shared memory accesses are equivalent
  - How can optimize shared memory accesses?
    - Moderately relevant questions

- To do justice to these topics we’ll need to talk first about scheduling threads for execution
  - Coming up next…
Execution Scheduling Issues
[NVIDIA cards specific]
Thread Execution Scheduling

- Topic we are about to discuss:
  - You launch on the device many blocks, each containing many threads
  - Several blocks can get executed simultaneously on one SM (8 SPs). How is this possible?
GeForce-8 Series HW Overview
Thread Scheduling/Execution

- Each Thread Block divided in 32-thread “Warps”
  - This is an implementation decision, not part of the CUDA programming model

- Warps are the basic scheduling unit in SM

- If 3 blocks are processed by an SM and each Block has 256 threads, how many Warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
Scheduling on the Hardware

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
- SM schedules and executes Warps that are ready to run
- As Thread Blocks complete kernel execution, resources are freed
  - SPA can launch next Block[s] in line
- NOTE: Two levels of scheduling:
  - For running [desirably] a large number of blocks on a small number of SMs (30/16/14/ etc.)
  - For running up to 24 (or 32, on Tesla C1060) warps of threads on the 8 SPs available on each SM
SM Warp Scheduling

- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp on C1060

- How is this relevant?
  - Suppose your code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:

\[
\frac{400}{(6 \times 4)} = 16.6667 \Rightarrow 17 \text{ Warps}
\]
Thread Blocks are Executed as Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block can execute independently
Organizing Threads into Warps

- **Thread IDs within a warp are consecutive and increasing**
  - This goes back to the 1D projection from thread index to thread ID
  - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.

- **Partitioning of threads in warps is always the same**
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- **While you can rely on ordering among threads, DO NOT rely on any ordering among warps since there is no such thing**
  - Warp scheduling is not something you control through CUDA
Thread and Warp Scheduling

- An SM can switch between warps with no apparent overhead
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all [active] threads execute the same instruction in lockstep fashion

\[ W_1 \rightarrow W_2 \rightarrow W_3 \rightarrow W_4 \rightarrow W_n \]

- Executing $W_n$
- Waiting for data
- Ready to execute

NVIDIA [J. Balfour]
Filling Warps

- Prefer thread block sizes that result in mostly full warps

**Bad:**  \( \text{kernel} \llll N, 1 \rrrr (\ldots) \)
**Okay:**  \( \text{kernel} \llll (N+31) / 32, 32 \rrrr (\ldots) \)
**Better:**  \( \text{kernel} \llll (N+127) / 128, 128 \rrrr (\ldots) \)

- Prefer to have enough threads per block to provide hardware with many warps to switch between

- This is how the GPU hides memory access latency

- Resource like `__shared__` may constrain number of threads per block
Control Flow Divergence

Consider the following code:

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads in the warp execute the `if` clause, the other half the `else` clause.
The system automatically handles control flow divergence conditions in which threads within a warp execute different paths through a kernel.

Often, this requires that the hardware executes multiple paths through a kernel for a warp.

- For example, both the if clause and the corresponding else clause.
Control Flow Divergence

[3/4]

```c
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        // g(x)
        t = g(x[i]);
    }
    else
    {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
```
Control Flow Divergence

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock

- In general, one does need to consider divergence when reasoning about the performance of a program
Performance of Divergent Code

- Performance decreases with degree of divergence in warps
- Here's an extreme example…

```c
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
    case 0 : x[i] = a(x[i]);
           break;
    case 1 : x[i] = b(x[i]);
           break;
    ...
    case 31: x[i] = v(x[i]);
           break;
    }
}
```
Performance of Divergent Code

- Compiler and hardware can detect when all threads in a warp branch in the same direction
  - For example, all take the `if` clause, or all take the `else` clause
  - The hardware is optimized to handle these cases without loss of performance
  - In other words, use of `if` or `switch` does not automatically translate into disaster:
    
    ```c
    if (threadIdx.x / WARP_SIZE >= 2) { }
    ```
    
    - Creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path. There is no warp divergence...

- The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
  - Avoids some branch divergence overheads, and is more efficient
  - Often acceptable performance with short conditional clauses
End of CUDA Basics

Issues Related to Improving Performance of CUDA Code
Memory Facts, Fermi GPUs

- There is 64 KB of fast memory on each SM that gets split between L1 cache and Shared Memory
  - You can split 64 KB as “L1/Sh: 16/48” or “L1/Sh: 48/16”

- L2 cache: 768 KB – one big pot available to *all* SMs on the device

- L1 and L2 cache used to cache accesses to
  - Local memory, including register spill
  - Global memory

- Whether reads are cached in [L1 & L2] or in [L2 only] can be partially configured on a per-access basis using modifiers to the load or store instruction
Fermi Memory Layout

[credits: NVIDIA]
## GPU-CPU Face Off

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050</th>
<th>CPU – Intel core i7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Cores</td>
<td>448</td>
<td>4 (8 threads)</td>
</tr>
<tr>
<td>Memory</td>
<td>64* KB L1, per SM 768 KB L2, all SMs 3 GB Device Mem.</td>
<td>- 32 KB L1 cache / core - 256 KB L2 (I&amp;D) cache / core - 8 MB L3 (I&amp;D) shared, all cores</td>
</tr>
<tr>
<td>Clock speed</td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>140 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>Floating point operations/s</td>
<td>515 x 10^9 Double Precision</td>
<td>70 x 10^9 Double Precision</td>
</tr>
</tbody>
</table>

* - split 48/16
More Memory Facts
[Fermi GPUs]

- All global memory accesses are cached

- A cache line is 128 bytes
  - It maps to a 128-byte aligned segment in device memory

- You can determine at *compile* time (through flags: `-dlicm=ca/cg`) if you double cache [L1 & L2] or only cache [L2 only]
  - If [L1 & L2], a memory access is serviced with a 128-byte memory transaction
  - If [L2 only], a memory access is serviced with a 32-byte memory transaction
    - This can reduce over-fetch in the case of scattered memory accesses
    - Good for irregular pattern access (sparse linear algebra)
More Memory Facts
[Fermi GPUs]

- If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently.

- The memory access schema is as follows:
  - Two memory requests, one for each half-warp, if the size is 8 bytes.
  - Four memory requests, one for each quarter-warp, if the size is 16 bytes.

- Each memory request is then broken down into cache line requests that are issued independently.

- NOTE: a cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.
How to Use L1 and L2

- Should you start programming to leverage L1 and L2 cache?

- The answer is: NO
  - GPU caches are not intended for the same use as CPU caches
    - Smaller sizes (on a per-thread basis, that is), not aimed at temporal reuse
      - Intended to smooth out some access patterns, help with spilled registers, etc.
  - Don’t try to block for L1/L2 like you would on CPU
    - You have 100s to 1000s of run-time scheduled threads hitting the caches
    - Instead of L1, you should start thinking how to leverage Shared Memory
      - Same bandwidth (they *physically* share the same memory)
      - Hardware will not evict behind your back

- Conclusions
  1. Optimize as if no caches were there
  2. The reason why we talk about this: it helps you understand when the GPU is good and when it’s not
Global Memory

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The **layout of the access** to global memory (the pattern of the access)
  - The **size/alignment** of the data you try to fetch from global memory
“Memory Access Layout”
What is it?

• The basic idea:
  • Suppose each thread in a warp accesses a global memory address for a load operation at some point in the execution of the kernel
  • These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place
  • Case (a) is called a “coalesced memory access”
    • If you end up with (b) this will adversely impact the overall program performance
  • Analogy
    • Can send one truck on six different trips to bring back each time a bundle of wood
    • Alternatively, can send truck to one place and get it back fully loaded with wood
Global Memory Access Schema
[Compute Capability 1.3]

- A global memory request for a warp is split in two memory requests, one for each half-warp.
- The following 5-stage protocol is used to determine the memory transactions necessary to service all threads in a half-warp.

**Stage 1:** Find the memory segment that contains the address requested by the lowest numbered active thread. The memory segment size depends on the size of the words accessed by the threads:
  - 32 bytes for 1-byte words,
  - 64 bytes for 2-byte words,
  - 128 bytes for 4-, 8- and 16-byte words.

**Stage 2:** Find all other active threads whose requested address lies in the same segment.

**Stage 3:** Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.

**Stage 4:** Carry out the transaction and mark the serviced threads as inactive.

**Stage 5:** Repeat until all threads in the half-warp are serviced.
Examples

[Preamble]

- Look at an example that deals with 32 bit words (4 bytes)
- This is the case when handling integers or floats
- Various scenarios are going to be considered to illustrate how the two factors (layout of access & alignment) come into play when accessing global memory
- Note that when handling 32 bit words, “segment size” represents 128 byte data chunks (all aligned at multiples of 128)
  - In what follows, a different color is associated with each 128 byte memory segment
  - In other words, two rows of the same color represent a 128-byte aligned segment
Example: Scenario 1

- Coalesced access in which all threads but one access the corresponding word in a segment

- This access pattern results in a single 64-byte transaction, indicated by the red rectangle

- Although one word is not requested, all data in the segment is fetched
  - Sometimes called an “over-fetch”

- If accesses by threads were permuted within this segment, still one 64-byte transaction would be performed on Tesla C1060
Example: Scenario 2

- Sequential threads in a half warp access memory that is sequential but not aligned with the segments.

- Given that the addresses fall within a 128-byte segment, a single 128-byte transaction is performed on Tesla C1060.
Example: Scenario 3

- A half warp accesses memory that is sequential but split across two 128-byte segments. Note that the request spans two different memory segments.

- On Tesla C1060, two transactions are performed: one 64-byte transaction and one 32-byte transaction result.
Example: Scenario 4

- Strided access to global memory, as shown in the code snippet below:

```c
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

- Although a stride of 2 above results in a single transaction, note that half the elements in the transaction are not used and represent wasted bandwidth.
Example: Scenario 4

[Cntd.]

- Strided access to global memory, as shown in the code snippet below:

```c
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

- As the stride increases, the effective bandwidth decreases until the point where 16 transactions are issued for the 16 threads in a half warp, as shown in the plot
Examples of Global Mem. Access by a Warp

- **Setup:**
  - You want to access floats or integers
  - In order words, each thread is requesting a 4-Byte word

- **Scenario A:** access is aligned and sequential
Examples of Global Mem. Access by a Warp

[Cntd.]

- **Scenario B: Aligned but non-sequential**

  ![Aligned and non-sequential diagram]

- **Scenario C: Misaligned and sequential**

  ![Misaligned and sequential diagram]
Why is this important?

- Compare Scenario B to Scenario C

- Basically, you have in Scenario C half the effective bandwidth you get in Scenario B
  - Just because of the alignment of your data access

- If your code is memory bound and dominated by this type of access, you might see a doubling of the run time...

- The moral of the story:
  - When you reach out to grab data from global memory, visualize how a full warp reaches out for access. Is the access coalesced and well aligned?
Think about this…

- Suppose you use in your program complex data constructs that could be organized using C-structures

- Based on what we’ve discussed so far today, how is it more advantageous to store data in global memory?
  - Alternative A: as an array of structures
  - Alternative B: as a structure of arrays
Technical Specifications and Features

[Short Detour]

This is us:
Fermi GPUs on Euler

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
“multiprocessor” stands for Stream Multiprocessor (what we called SM)
CUDA Optimization: Execution Configuration Heuristics
Blocks per Grid Heuristics

- # of blocks > # of stream multiprocessors (SMs)
  - If this is violated, then you’ll have idling SMs

- # of blocks / # SMs > 2
  - Multiple blocks can run concurrently on a multiprocessor
  - Blocks that aren’t waiting at a `__syncthreads()` keep the hardware busy
  - Can do this subject to resource availability – registers, shared memory
  - NOTE: the block scheduler never assigns more than 8 blocks to one SM (hardware constraint)

- # of blocks > 100 to scale to future devices
  - Blocks waiting to be executed in pipeline fashion
  - To be on the safe side, 1000’s of blocks per grid will scale across multiple generations
  - If you bend backwards to meet this requirement maybe GPU not the right choice
Threads Per Block Heuristics

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough registers to compile and invoke successfully
  - This all depends on your computation, so experiment!

- Always use the nvvp profiler to understand how many registers you used, what bandwidth you reached, etc.
Occupancy

- In CUDA, executing other warps is the only way to hide latencies and keep the hardware busy.

- Occupancy = Number of warps running concurrently on a SM divided by maximum number of warps that can run concurrently.
  - When adding up the number of warps, they can belong to different blocks.

- Can have up to 48 warps managed by one Fermi SM.
  - For 100% occupancy your application should run with 48 warps on an SM.

- Many times one can’t get 48 warps going due to hardware constraints.
CUDA Optimization: A Balancing Act

- **Hardware constraints:**
  - Number of registers per kernel
    - 32K per multiprocessor, partitioned among concurrent threads active on the SM
  - Amount of shared memory
    - 16 or 48 KB per multiprocessor, partitioned among SM concurrent blocks

- **Use** `-maxrregcount=N` **flag on** `nvcc`
  - **N** = desired maximum registers / kernel
  - At some point “spilling” into local memory may occur
    - Might not be that bad, there is L1 cache that helps to some extent

- Recall that you cannot have more than 8 blocks executed by one SM
CUDA GPU Occupancy Calculator

1. Select Compute Capability (click) [2.0]
2. Enter your resource usage:
   - Threads Per Block
   - Registers Per Thread
   - Shared Memory Per Block (bytes)
   - Occupancy per Multiprocessor
   - Occupancy per Multiprocessor

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Varying Block Size

Varying Register Count

Varying Shared Memory Usage

CUDA Occupancy Calculator

Version 2.1

Google: "cuda occupancy calculator"
Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize your solution.

2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits.

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU.

Writing CUDA Software: High-Priority Recommendations

4. Ensure global memory accesses are coalesced whenever possible.

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution).

6. Avoid different execution paths within the same warp.

Writing CUDA Software: Medium-Priority Recommendations

1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts.

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy).

3. The number of threads per block should be a multiple of 32 threads because this provides optimal computing efficiency and facilitates coalescing.

4. Use the fast math library whenever speed is important and you can live with a tiny loss of accuracy

5. Prefer faster, more specialized math functions over slower, more general ones when possible

Writing CUDA Software: Low-Priority Recommendations

1. For kernels with long argument lists, place some arguments into constant memory to save shared memory

2. Use shift operations to avoid expensive division and modulo calculations

3. Avoid automatic conversion of doubles to floats

4. Make it easy for the compiler to use branch predication in lieu of loops or control statements

The Common Pattern to CUDA Programming

- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU
- **Phase 2**: GPU crunches numbers based on the kernel you defined
- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up…)
The Common Pattern to CUDA Programming

- **Phase 1:** Allocate memory on the device and copy to the device the data required to carry out computation on the GPU
- **Phase 2:** GPU crunches numbers based on the kernel you defined
- **Phase 3:** Bring back the results from the GPU. Free memory on the device (clean up…)

**Rules of Thumb for Efficient GPU Computing:**
1. Get the data on the GPU and keep it there
2. Give the GPU enough work to do
3. Focus on data reuse within the GPU to avoid memory bandwidth limitations